Lattice QCD GPU programming with Quda and Grid

October 24, 2025

renwick.james.hudspith@gmail.com

Outline of the series

- Lecture 1 - Introduction to GPUs and CUDA parallelism
- Lecture 2 - Writing interfaces and kernels in QUDA, intro to cuBLAS
- Lecture 3 - BLAS in QUDA and GRID GPU parallelism

Opinions expressed here and in following lectures are mine alone as are mistakes

Outline of this lecture

- What is a GPU?
- What is the parallelism strategy?
  - What is the hierarchy of GPU hardware?
- Example CUDA codes
  - Blocks and Threads - vector addition
  - Reducing an array

Quda strategy *is* CUDA parallelism

Consulting Wikipedia: "A Graphics Processing Unit (GPU) is a specific piece of hardware for image processing and computer graphics."

They can be attached via PCIE as a separate graphics card or integrated on the motherboard as we have in smartphones, laptops, ... etc.

It turns out that the same demands for digital image processing are comparable to those in High Performance Computing (HPC), provided your problem can be computed in an embarrassingly parallel fashion.

The Atari 2600 (1977) had a dedicated graphics chip the Television Interface Adapter (TIA) as well as a 1.19 MHz microprocessor for game logic.

It is the demand for better graphics in video games, more realistic renders, combined with Moore's law that has led us to where we are today. The current fad for AI and LLMs will do the same.

Image By Evan-Amos - Own work, Public Domain,

https://commons.wikimedia.org/w/index.php?curid=38731073

3dfx Voodoo, the 90s, and consumer hardware

Away from cabinets and bespoke hardware interfaces to televisions the 90s was the era of the home desktop PC and a revolution in home gaming. The consumer demand was for 3d graphics due to the advent of games such as Quake. The term GPU was apparently coined by SONY for the Playstation 1.

The market leader for home desktops at this time was probably 3dfx, with the introduction of the Voodoo series (late 90s) of graphics cards specifically for rendering 3D graphics. 3D graphics are very compute-intensive and this is where the drive comes for future HPC uses.

By the early 2000s more companies entered the market, Nvidia released the geForce series in late 1999 as well as ATI Radeon (acquired by AMD in 2006 for $5.4 billion).

None of these are the multi-core behemoths we think of today but still bespoke interfaces for a specific niche. These are all MFLOP cards.

A step into my past - the 9800 GT

CUDA (Compute Unified Device Architecture) released in Feb 2007, gives a proprietary interface to allow simplified, general parallel computing on Nvidia devices.

The 9800 GT (2008) was the first Graphics card I ran Quda with in the very early 2010s, this is the hardware I had:

- 112 "shading units" or CUDA cores, 1SM
- 512 MB GDDR3 RAM
- 600 MHZ graphics clock boost to 1500
- PCIE
- 465 GFlops (single precision) 75 Watts

RAM was small. CPU clock speeds were low. Power draw was high. Single precision was good enough for physics in games. For general purpose HPC this wasn't competitive with a Blue Gene L/P/Q


EN9800GT HybridPower
ASUS
Rock Solid • Heart Touching
ASUS Exclusive
3
512MB
DDR3
PCI EXPRESS 2.0
HDMI
EN9800GT HybridPower

What we use on Perlmutter

What we have been hosing for the last year on Perlmutter are 4xA100 (2020) GPUS per node on a single AMD EPYC 7763 64-core CPU-node. A single A100 has the following specs:

- 6912 processor cores 108 SMs
- **40 GB HBM2 RAM**
- 765 MHZ graphics clock boost to 1450
- **SXM** (PCIE)
- 312 (half) **156 (TF32)** 19.5 (single) 9.7 (double) TFlops 400 Watts

As transistor sizes have dropped and fabrication has improved, we can pack more on a die at comparable power draw. $\approx 340\times$ compute over the 9800GT for $\approx 5\times$ the power draw.
The tensor cores have so much compute for deep neural networks as models do not need high precision for their weight calculations.

GPU parallel

We have seen that core clock speeds have practically stayed the same for GPUs (to stay in line with typical RAM clock speeds) for more than 15 years but the core count has ballooned. This is also what we see in consumer CPUs - to stay in line with Moore's law companies have moved to a multi-core processor - *GPU performance has outstripped CPU*

A major take-home GPU parallelism *is* Single Instruction Multiple Thread (SIMT) parallelism - an instruction is broadcast to multiple hardware threads that are synchronised (well, maybe not anymore) and they all execute this same instruction on different parts of data. A downside of this is the Instruction Latency is long on a GPU, but that is somewhat hidden from us.

For Nvidia a "Warp" sends the instruction to 32 hardware threads. AMD calls this a "Wavefront". In CUDA the kernel operates on a grid of Warps.

SPMD vs SIMT

On the CPU-side we are likely familiar with OpenMP which follows the Single Program Multiple Data (SPMD) paradigm, a comparison of the two is below.

- Each thread can do it's own thing including branching
- All threads can access global program data
- For vanilla "parallel for" workload is shared equally between threads

- Each thread in warp is doing exactly the same thing (SIMD)
- Shared device L2 Cache and fast local L1
- Workload is shared evenly between threads in a warp you choose blocking

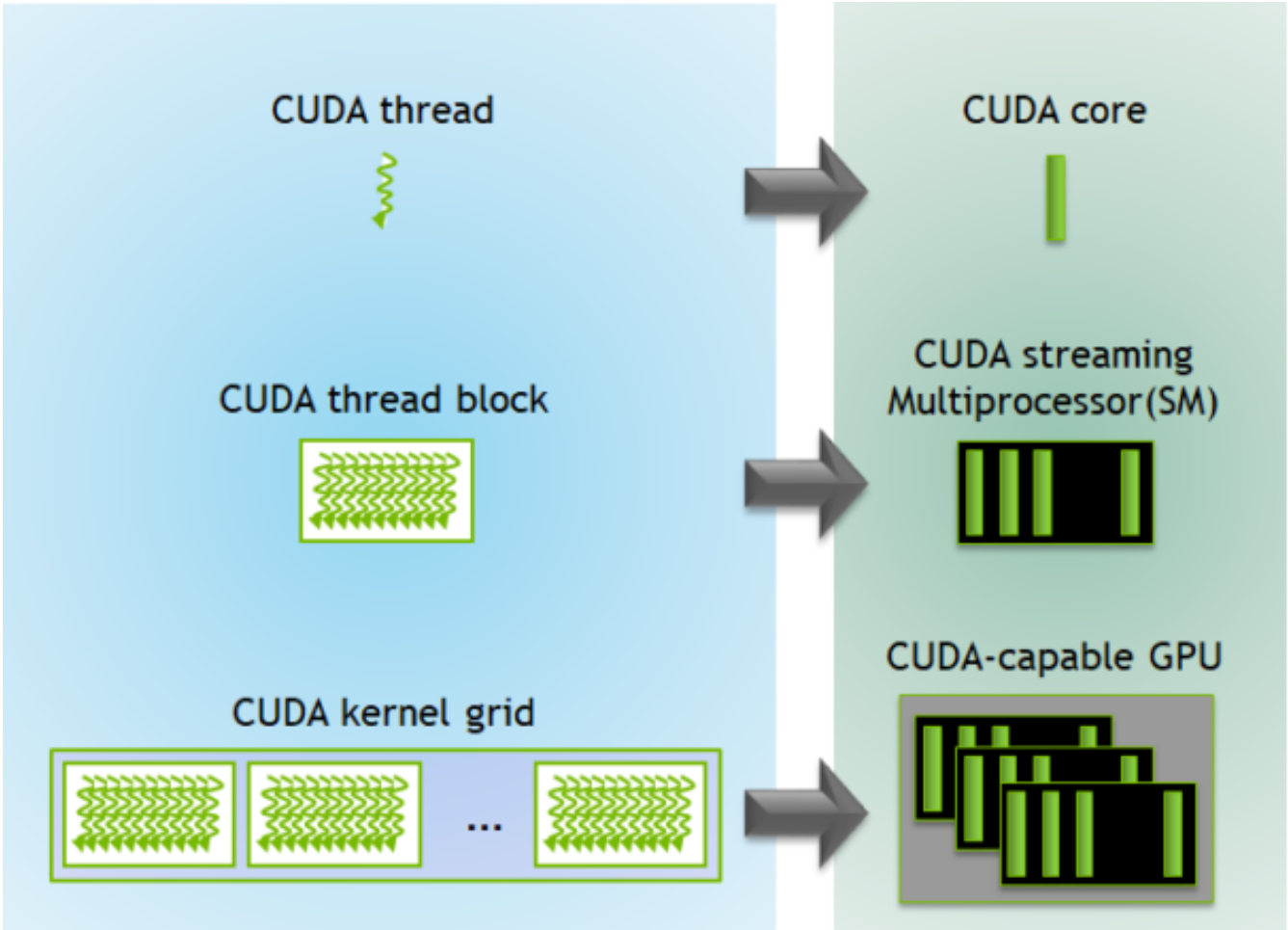
If we restrict OpenMP's workload it really looks like that for a SIMT GPU both exploit data parallelism. This is the root of why we can offload OpenMP loops to the GPU in v5.0, and the crux of Grid's parallelism strategy.

Introducing CUDA

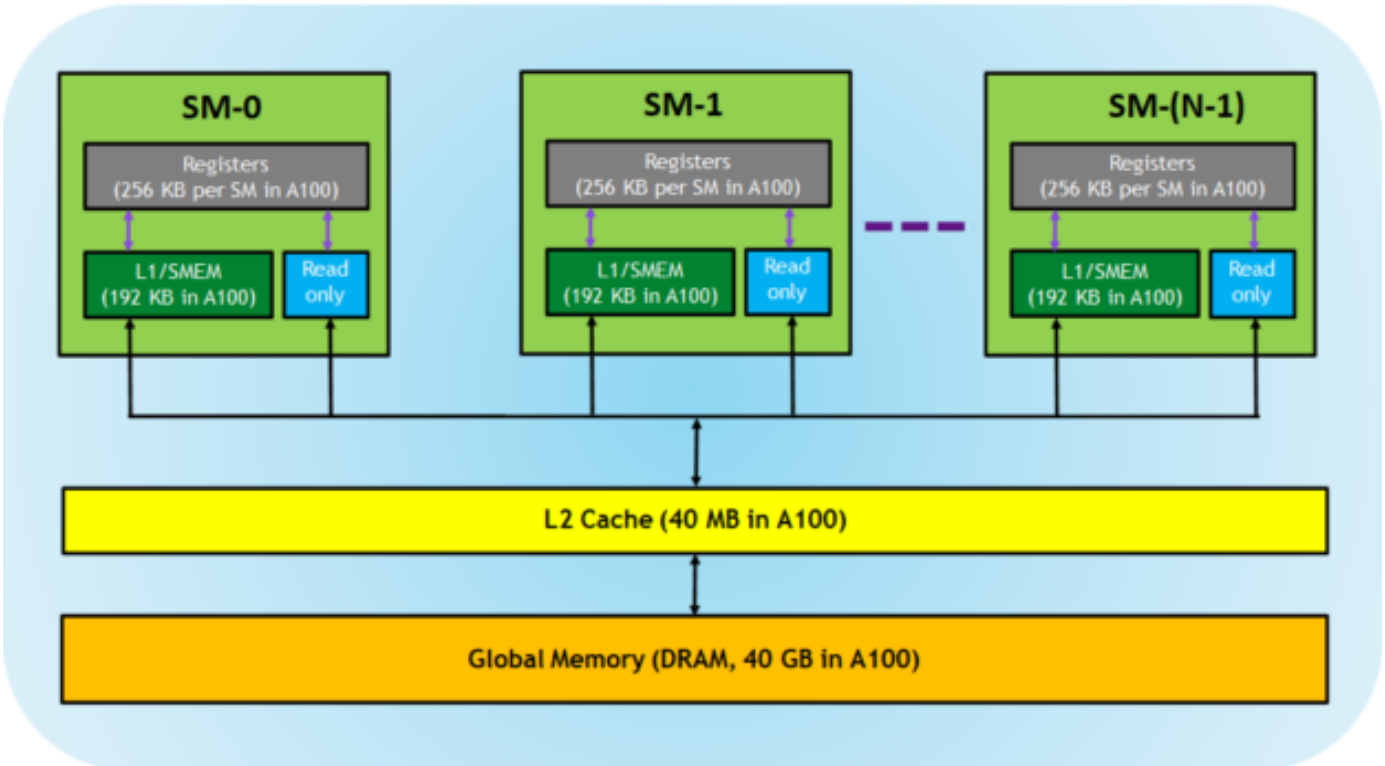
Ok so now we introduce the hierarchy of what CUDA operates on.
Each GPU is built from some number of Streaming Multiprocessors (SMs). These are like a multi-core CPU with small L1 cache and simple instruction set.

1. A thread uses an individual CUDA core
2. A Warp is a collection of 32 threads
3. A threadblock is a collection (up to 32 Warps) that will execute on a single SM

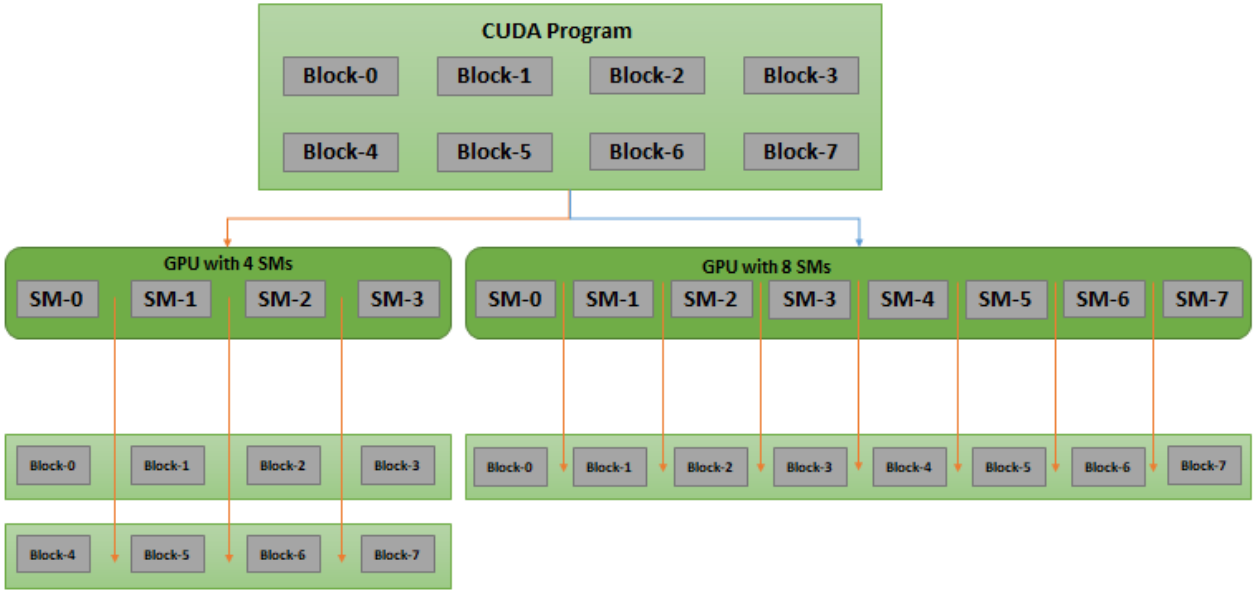
If you want to run an instruction that uses only 24 threads then all 32 in the warp will be used and the result of the last 8 will be thrown away (or try and access out of bounds memory!).
All of the threads (max 1024) in the same threadblock will execute on the same SM
If you have more threadblocks than will fit on your SMs they will be scheduled when available



https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/



https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/



(Left) If I have a GPU with 4 SMs and I have a job that uses 8 threadblocks CUDA will automatically execute the first 4 blocks and then the next four on the GPU. If I have the same code executing on the GPU on the right with 8 SMs each one will be given a threadblock. CUDA will deal with different architectures.

https://developer.nvidia.com/blog/cuda-refresher-getting-started-with-cuda/

An example from the intro to CUDA

CUDA is a c API that allows for General Purpose GPU programming
Let's say we want to add a vector of integers using CUDA, this is the kernel we can run on the device

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

Here we choose to parallelise over the 1D blockIdx, so each thread block uses only 1 thread
We need to include the basic c headers plus those from cuda and compile with nvcc

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <cuda.h>
#include <cuda_runtime.h>
```

We must copy onto the device and off the device (cudaMallocHost likely better!)

```
#define N 512
int main(void) {
    int *a, *b, *c; // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);
    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);
    // Alloc space for host copies of a, b, c
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size)
```

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);
    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Here the line `add<<<N,1>>>` is very important. It says I want N threadblocks and 1 thread per block. This is the kernel we are launching.

From what we know about the GPU architecture this is a silly choice and we should just run on threads instead and make use of the fast L1 cache. The change is simple

```
__global__ void add(int *a, int *b, int *c) {
   c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

And the launch also is changed

```
add<<<1,N>>>(d_a, d_b, d_c);
```

This will all run more efficiently on 1 SM provided we don't run out of threads. And is indicative of the fine-grained parallelism available from the CUDA API. We can also do a mixture of blocks and threads for 2D/3D parallelism, which is convenient for matrices and 3D graphics.

A CUDA example from the developer blog - Matrix addition

A kernel that adds two matrices on the device. Assuming N is #defined in the code somewhere above

```
// Kernel - Adding two matrices MatA and MatB
__global__ void MatAdd(float MatA[N][N], float MatB[N][N],
float MatC[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        MatC[i][j] = MatA[i][j] + MatB[i][j];
}
```

Note each block has an x and y index, i.e. thread block is a 2D index of threads. Actually can be up to a 3D one provided $x.y.z \leq 1024$ and $z \leq 64$.

The calling code expects the matrices MatA and MatB to be set somewhere and copied to the device, now using the CUDA dim3 arguments (just a 3-vector of ints).

```
int main()
{
.... Copies to the device .....
// Matrix addition kernel launch from host code
dim3 threadsPerBlock(16, 16);
dim3 numBlocks((N+threadsPerBlock.x-1)/threadsPerBlock.x,
               (N+threadsPerBlock.y-1)/threadsPerBlock.y);
MatAdd<<<numBlocks, threadsPerBlock>>>(MatA, MatB, MatC);
.... Copies back ......
}
```

We have 16x16=256 threads per threadblock (less than the CUDA limit of 1024). The number of blocks is worked out by the size of the matrix. dim3 sets dimension to 1 that isn't an argument.

Getting used to CUDA parallelism

Can you take the 1D vector addition code and change it to floating point and generalise it to use both blocks and threads?

- For a large number of vector indices (say N=32*1024*1024) what is the optimal block/thread combination?
- How does this tuning compare to copies to and from the device?

Compare to vadd.cu for my code in this directory

Reductions in CUDA

Another common operation we have is a reduction - i.e. a sum of all numbers so let's try that.

The idea is to sum over threads in local memory and put those mini-sums into a global array on the device. We then copy that back to the host and finish the sum. Why? We cannot synchronize blocks, nor would we want to so we just let them accumulate.

A post about optimising a reduction is here:
https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
and we will pick that up at the third stage "sequential addressing"

The idea is that the inner loop "s" splits the memory into 2 and adds them overwriting the lower half. It then does the same for the lower half .. and so on until we have summed down to element 1. We cannot just do a for loop over threads.

```
__global__ void vsum(float *out, float *a, int n){
    extern __shared__ float sdata[] ; // shared memory
    int tid = threadIdx.x ;
    int idx = blockIdx.x*blockDim.x+tid ;
    sdata[tid] = a[idx] ; // global to local
    __syncthreads() ;
    for( int s = blockDim.x/2 ; s > 0 ; s>>=1 ) {
      if( tid < s ) {
          sdata[tid] += sdata[tid+s] ;
      }
      __syncthreads() ;
    }
    if( tid == 0 ) out[blockIdx.x] = sdata[0] ;
}
```

We know that we have a SIMD warp of 32 threads so unrolling the loop

```
__device__ inline void wred( volatile float *sdata ,
                             const int tid ){
   sdata[tid] += sdata[tid+32] ;
   sdata[tid] += sdata[tid+16] ;
   sdata[tid] += sdata[tid+8] ;
   sdata[tid] += sdata[tid+4] ;
   sdata[tid] += sdata[tid+2] ;
   sdata[tid] += sdata[tid+1] ;
}
```

The for loop changes and we call the unrolled guy without synch-ing

```
for( int s = blockDim.x/2 ; s > 32 ; s>>=1 ) {
     .....
}
if( tid < 32 ) wred( sdata , tid ) ;
if( tid == 0 ) out[blockIdx.x] = sdata[0] ;
```

To Summarise

GPU architecture hasn't always been useful for us in HPC. But now it really is, typically for low precision. This hasn't been dictated by us (scientists, people running Molecular Dynamics) particularly, but by the market for video games, video rendering, crypto mining, AI hype ... etc. We are very much now in the age of GPU dominance though.

Due to the fine-grained parallelism of the GPU we have many ways to skin this particular cat, this implies the need for auto-tuning as each piece of hardware will have different optimal parallelism strategies. Some operations are quite different to how we would normally parallelise, like a reduction, now look at vsum.cu in this directory. Quda can do some of this for us if we want.

Quda (as we will see next lecture) approaches GPU parallelism in a much more CUDA-like fashion, GRID in a more OpenMP like way (lecture 3).